JExtract: An Eclipse Plug-in for Recommending Automated Extract Method Refactorings

Danilo Silva[1], **Ricardo Terra**[2], **Marco Túlio Valente**[1]

[1]Federal University of Minas Gerais, Brazil

[2]Federal University of Lavras, Brazil

{danilofs,mtov}@dcc.ufmg.br, terra@dcc.ufla.br

Abstract. *Although Extract Method is a key refactoring for improving program comprehension, refactoring tools for such purpose are often underused. To address this shortcoming, we present JExtract, a recommendation system based on structural similarity that identifies Extract Method refactoring opportunities that are directly automated by IDE-based refactoring tools. Our evaluation suggests that JExtract is more effective (w.r.t. recall and precision) to identify contiguous misplaced code in methods than JDeodorant, a state-of-the-art tool.*

Tool demonstration video. *http://youtu.be/6htJOzXwRNA*

1. Introduction

Refactoring has increased in importance as a technique for improving the design of existing code [2], e.g., to increase cohesion, decrease coupling, foster maintainability, etc. Particularly, Extract Method is a key refactoring for improving program comprehension. Besides promoting reuse and reducing code duplication, it contributes to readability and comprehensibility, by encouraging the extraction of self-documenting methods [2].

Nevertheless, recent empirical research indicate that, while Extract Method is one of the most common refactorings, automated tools supporting this refactoring are most of the times underused [5, 4]. For example, Negara et al. found that Extract Method is the third most frequent refactoring, but the number of developers who apply the refactoring manually is higher than the number of those who do it automatically [5]. Moreover, current tools focus only on automating refactoring application, but developers expend considerable effort on the manual identification of refactoring opportunities.

To address this shortcoming, this paper presents `JExtract`, a tool that implements a novel approach for recommending automated Extract Method refactorings. The tool was designed as a plug-in for the Eclipse IDE that automatically identifies, ranks, and applies the refactoring when requested. Thereupon, `JExtract` may aid developers to find refactoring opportunities and contribute to a widespread adoption of refactoring practices. The underlying technique is inspired by the *separation of concerns* design guideline. More specifically, we assume that *the structural dependencies established by Extract Method candidates should be very different from the ones established by the remaining statements in the original method*.

The remainder of this paper is structured as follows. Section 2 describes the `JExtract` tool, including its design and implementation. Section 3 discusses related tools and Section 4 presents final remarks.

2. The `JExtract` tool

`JExtract` is a tool that analyzes the source code of methods and recommends Extract Method refactoring opportunities, as illustrated in Figure 1. First, the tool generates all Extract Method possibilities for each method. Second, these possibilities are ranked according to a scoring function based on the similarity between sets of dependencies established in the code.

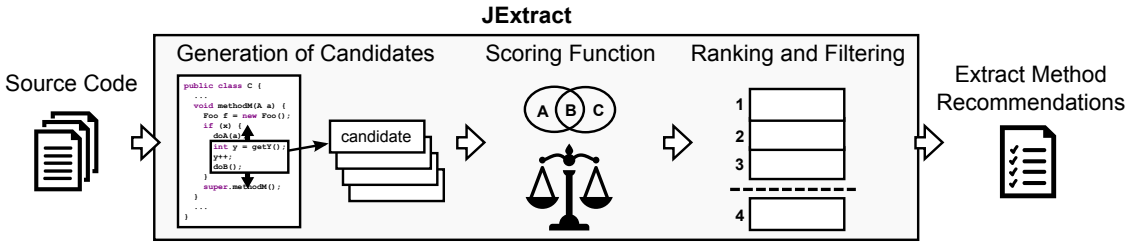

Figure 1. The `JExtract` tool

This main section of the paper is organized as follows. Subsection 2.1 provides an overview of our approach for identifying Extract Method refactoring opportunities. Subsection 2.2 describes the design and implementation of the tool. Finally, Subsection 2.3 presents the results of our evaluation in open-source systems. A detailed description of the recommendation technique behind JExtract is present in a recent full technical paper [9].

2.1. Proposed Approach

The approach is divided in three phases: *Generation of Candidates*, *Scoring*, and *Ranking*.

2.1.1. Generation of candidates

This phase is responsible for identifying all possible Extract Method refactoring opportunities. First, we split the methods into blocks, which consist of sequential statements that follow a linear control flow. As an example, Figure 2 presents method `mouseRelease` of class `SelectionClassifierBox`, extracted from ArgoUML. We can notice that each statement is labeled using the `SX.Y` pattern, where `X` and `Y` denote the block and the statement, respectively. For example, `S2.3` is the third statement of the second block, which declares a variable `cw`.

```
public void mouseReleased(MouseEvent me) {
S1.1  for (Button btn : buttons) {
S2.1    int cx = btn.fig.getX() + btn.fig.getWidth() - btn.icon.getIconWidth();
S2.2    int cy = btn.fig.getY();
S2.3    int cw = btn.icon.getIconWidth();
S2.4    int ch = btn.icon.getIconHeight();
S2.5    Rectangle rect = new Rectangle(cx, cy, cw, ch);
S2.6    if (rect.contains(me.getX(), me.getY())) {
S3.1      Object metaType = btn.metaType;
S3.2      FigClassifierBox fcb = (FigClassifierBox) getContent();
S3.3      FigCompartment fc = fcb.getCompartment(metaType);
S3.4      fc.setEditOnRedraw(true);
S3.5      fc.createModelElement();
S3.6      me.consume();
S3.7      return;
        }
    }
S1.2  super.mouseReleased(me);
}
```

Figure 2. An Extract Method candidate in a method of ArgoUML (`S3.2` to `S3.5`)

Second, we generate all Extract Method candidates based on Algorithm 1 (extracted from [9]).

Algorithm 1 Candidates generation algorithm [9]

Input: A method M
Output: List with Extract Method candidates

```
1:  Candidates ← ∅
2:  for all block B ∈ M do
3:      n ← statements(B)
4:      for i ← 1, n do
5:          for j ← i, n do
6:              C ← subset(B, i, j)
7:              if isValid(C) then
8:                  Candidates ← Candidates + C
9:              end if
10:         end for
11:     end for
12: end for
```

Fundamentally, the three nested loops in Algorithm 1 (lines 2, 4, and 5) enforce that the list of selected statements attend the following preconditions:

- Only continuous statements inside a block are selected. In Figure 2, for example, it is not possible to select a candidate with `S3.2` and `S3.4` without including `S3.3`.
- The selected statements are part of a single block of statements. In Figure 2, for example, it is not possible to generate a candidate with both `S2.6` and `S3.1` since they belong to distinct blocks.
- When a statement is selected, the respective children statements are also included. In Figure 2, for example, when statement `S2.6` is selected, its children statements `S3.1` to `S3.7` are also included.

Last but not least, we do not ensure that every iteration of the loop yields an Extract Method candidate because: (i) a candidate recommendation must respect a size threshold defined by parameter *Minimum Extracted Statements*. The value is preset to 3 (changeable), which means that an Extract Method candidate has to have at least three statements; and (ii) a candidate recommendation must respect the preconditions defined by the Extract Method refactoring engine.

2.1.2. Scoring

This phase is responsible for scoring the possible Extract Method refactoring opportunities generated in the previous phase, using a technique inspired by a Move Method recommendation heuristic [7]. Assume m' as the selection of statements of an Extract Method candidate and m'' the remaining statements in the original method m. The proposed heuristic aims to minimize the structural similarity between m' and m''.

Structural Dependencies: The set of dependencies established by a selection of statements S with variables, types, and packages is denoted by $Dep_{var}(S)$, $Dep_{type}(S)$, and $Dep_{pack}(S)$, respectively. These sets are constructed as described next.

- *Variables:* If a statement s from a selection of statements S declares, assigns, or reads a variable v, then v is added to $Dep_{var}(S)$. In a similar way, reads from and writes to formal parameters and fields are considered.

- *Types:* If a statement s from a selection of statements S uses a type (class or interface) T, then T is added to $Dep_{type}(S)$.
- *Packages:* For each type T included in $Dep_{type}(S)$, as described in the previous item, the package where T is implemented and all its parent packages are also included in $Dep_{pack}(S)$.

For instance, assume m' as the highlighted code in Figure 2 (i.e., an Extract Method candidate) and m'' the remaining statements in the original method `mouseReleased`. On one hand, $Dep_{var}(m') = \{\texttt{metaType}, \texttt{fc}, \texttt{fcb}\}$. On the other hand, the set $Dep_{var}(m'') = \{\texttt{metaType}, \texttt{btn}, \texttt{cy}, \texttt{cx}, \texttt{cw}, \texttt{ch}, \texttt{buttons}, \texttt{me}, \texttt{rect}\}$. In this case, the intersection between these two sets contains only `metaType`. Moreover, the computation of `fc` and `fcb` is isolated from the remaining code. Therefore, one can claim that m' is cohesive and decoupled from m'', i.e., a good separation of concerns is achieved.

Scoring Function: To compute the dissimilarity between m' and m'', we rely on the distance between the dependency sets Dep' and Dep'' using the Kulczynski similarity coefficient [10, 7]:

$$dist(Dep', Dep'') = 1 - \frac{1}{2}\Big[\frac{a}{(a+b)} + \frac{a}{(a+c)}\Big]$$

where $a = |Dep' \bigcap Dep''|$, $b = |Dep' \setminus Dep''|$, and $c = |Dep'' \setminus Dep'|$.

Thus, let m' be the selection of statements of an Extract Method candidate for method m. Let also m'' be the remaining statements in m. The score of m' is defined as:

$$\begin{aligned} score(m') = \; & 1/3 \times dist(Dep_{var}(m'), Dep_{var}(m'')) \; + \\ & 1/3 \times dist(Dep_{type}(m'), Dep_{type}(m'')) \; + \\ & 1/3 \times dist(Dep_{pack}(m'), Dep_{pack}(m'')) \end{aligned}$$

The scoring function is centered on the observation that a good Extract Method candidate should encapsulate the use of variables, types, and packages. In other words, we should maximize the distance between the dependency sets Dep' and Dep''.

2.1.3. Ranking

This phase is responsible for ranking and filtering the Extract Method candidates based on the score computed in the previous phase. Basically, we sort the candidates and filter them according to the following parameters: (i) *Maximum Recommendations per Method*. The value is preset to 3 (changeable), which means that the tool triggers up to three recommendations for each method; and (ii) *Minimum Score Value*, which has to be configured when the user desires to setup a minimum dissimilarity threshold.

2.2. Internal Architecture and Interface

We implemented `JExtract` as a plug-in on top of the Eclipse platform. Therefore, we rely mainly on native Eclipse APIs, such as Java Development Tools (JDT) and Language

Toolkit (LTK). The current `JExtract` implementation follows an architecture with five main modules:

1. *Code Analyzer*: This module provides the following services to other modules: (a) it builds the structure of block and statements (refer to Subsection 2.1.1); (b) it extracts the structural dependencies (refer to Subsection 2.1.2); and (c) it checks if an Extract Method candidate satisfies the underlying Eclipse Extract Method refactoring preconditions. In fact, this module contains most communication between `JExtract` and Eclipse APIs (e.g., `org.eclipse.jdt.core` and `org.eclipse.ltk.core.refactoring`).

2. *Candidate Generator*: This module generates all Extract Method candidates based on Algorithm 1 and hence depends on service (a) of module *Code Analyzer*.

3. *Scorer*: This module calculates the dissimilarity of the Extract Method candidates generated by module *Candidate Generator* (refer to Subsection 2.1.2) and hence depends on service (b) of module *Code Analyzer*.

4. *Ranker*: This module ranks and filters the Extract Method candidates generated by module *Candidate Generator* and scored by module *Scorer*. It depends on service (c) of module *Code Analyzer* to filter candidates not satisfying preconditions.

5. *UI*: This module consists of the front-end of the tool, which relies on the Eclipse UI API (`org.eclipse.ui`) to implement two menu extensions, six actions, and one main view. Moreover, it depends on module UI from LTK (`org.eclipse.ltk.ui.refactoring`) to delegate the refactoring appliance to the underlying Eclipse Extract Method refactoring tool.

Such architecture permits the extension of our tool. For example, the *Scorer* module may be replaced by one that employs a new heuristic based on semantic and structural information. As another example, the *Candidate Generator* module may be extended to support the identification of non-contiguous code fragments.

Figure 3 presents `JExtract`'s UI, displaying method `mouseReleased` previously presented in Figure 2. When a developer triggers `JExtract` to identify Extract Method refactoring opportunities for this method, it opens the *Extract Method Recommendations* view to report the potential recommendations. In this case, the best candidate consists of the extraction of statements `S3.2` to `S3.5` whose dissimilarity score is 0.7148.

2.3. Evaluation

We conducted two different but complementary empirical studies.

Study #1: In our previous paper [9], we evaluated the recommendations provided by our tool on three systems to assess precision and recall. We extended this study to consider minor modifications to the ranking method and to compare the results with `JDeodorant`, a state-of-the-art tool that identifies Extract Method opportunities [11]. For each system S, we apply random Inline Method refactoring operations to obtain a modified version S'.

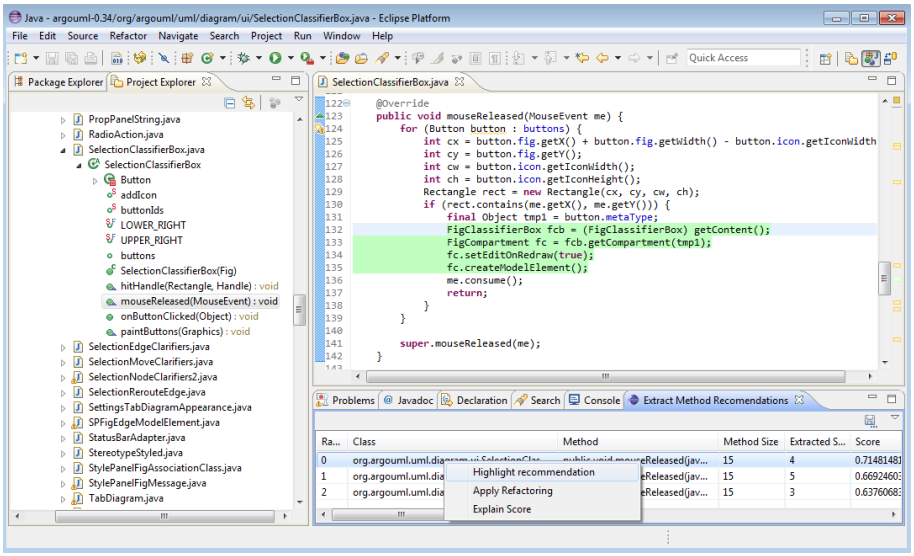

Figure 3. JExtract UI

We assume that good Extract Method opportunities are the ones that revert the modifications (i.e., restoring S from S').

Table 1. Study #1 – Recall and precision results

		JExtract						JDeodorant	
		Top-1		Top-2		Top-3			
System	#	Recall	Prec.	Recall	Prec.	Recall	Prec.	Recall	Prec.
JHotDraw 5.2	56	19 (34%)	34%	26 (46%)	24%	32 (57%)	20%	2 (4%)	5%
JUnit 3.8	25	13 (52%)	52%	16 (64%)	33%	18 (72%)	25%	0 (0%)	0%
MyWebMarket	14	12 (86%)	86%	14 (100%)	50%	14 (100%)	33%	2 (14%)	33%
Total	***95***	***44 (46%)***	46%	***56 (59%)***	***30%***	***64 (67%)***	***23%***	***4 (4%)***	***6%***

Table 1 reports recall and precision values achieved using `JExtract` with three different configurations (*Top-k Recommendations per Method*). While a high parameter value favors recall (e.g., Top-3), a low one favors precision (e.g., Top-1). Table 1 also presents results achieved using `JDeodorant` with its default settings. As the main finding, `JExtract` outperforms `JDeodorant` regardless of the configuration used.

Study #2: We replicate the previous study in other ten popular open-source Java systems to assess how the precision and recall rates would vary. Nevertheless, we do not compare our results with `JDeodorant` since we were not able to reliably provide the source code of all required libraries, as demanded by JDeodorant.

Table 2 reports the recall and precision values achieved using the same settings from the previous study. On one hand, the overall recall value ranges from 25% to 49.2%. On the other hand, the overall precision value ranges from 25% to 16.7%. We argue these values are acceptable for two reasons: (i) we only consider as correct a recommendation that matches exactly the one at the oracle; thus, a slight difference of including (or excluding) a statement is enough to be considered a miss; and (ii) the modified methods may have preexisting Extract Method opportunities, besides the ones we introduced, that will be considered wrong by our oracle.

Table 2. Study #2 – Recall and precision results

		JExtract					
		Top-1		Top-2		Top-3	
System	#	Recall	Prec.	Recall	Prec.	Recall	Prec.
Ant 1.8.2	964	235 (24.4%)	24.4%	363 (37.7%)	19.1%	460 (47.7%)	16.3%
ArgoUML 0.34	439	98 (22.3%)	22.3%	160 (36.4%)	18.3%	186 (42.4%)	14.4%
Checkstyle 5.6	533	227 (42.6%)	42.6%	338 (63.4%)	31.9%	389 (73.0%)	24.7%
FindBugs 1.3.9	714	179 (25.1%)	25.1%	278 (38.9%)	19.7%	350 (49.0%)	16.7%
FreeMind 0.9.0	348	85 (24.4%)	24.4%	134 (38.5%)	19.4%	181 (52.0%)	17.8%
JFreeChart 1.0.13	1,090	204 (18.7%)	18.7%	396 (36.3%)	18.2%	536 (49.2%)	16.5%
JUnit 4.10	35	11 (31.4%)	32.4%	17 (48.6%)	26.6%	22 (62.9%)	23.7%
Quartz 1.8.3	239	99 (41.4%)	41.4%	125 (52.3%)	26.5%	142 (59.4%)	20.4%
SQuirreL SQL 3.1.2	39	15 (38.5%)	38.5%	18 (46.2%)	23.7%	20 (51.3%)	18.2%
Tomcat 7.0.2	1,076	214 (19.9%)	19.9%	325 (30.2%)	15.2%	409 (38.0%)	12.8%
Total	***5,477***	***1,367 (25.0%)***	***25.0%***	***2,154 (39.3%)***	***19.8%***	***2,695 (49.2%)***	***16.7%***

3. Related Tools

Recent empirical research shows that automated refactoring tools, especially those supporting Extract Method refactorings, are most of the times underused [5, 4]. In view of such circumstances, recent studies on identification of refactoring opportunities are seeking to address this shortcoming. In this paper, we implemented our approach in a way that it can be straightforwardly incorporated to the current development process through a tool that identifies, ranks, and automate Extract Method refactoring opportunities [9].

`JMove` is the refactoring recommendation system our approach is inspired by [7, 6]. The tool identifies Move Method refactoring opportunities based on the similarity between dependency sets [7]. More specifically, it computes the similarity of the set of dependencies established by a given method m with (i) the methods of its own class C_1 and (ii) the methods in other classes of the system ($C_2, C_3, ..., C_n$). Whereas JMove recommends moving a method m to a more similar class C_i, our current approach recommends extracting a fragment from a given method m into a new method m' when there is a high dissimilarity between m' and the remainder statements in m.

`JDeodorant` is the state-of-the-art system to identify and apply common refactoring operations in Java systems, including Extract Method [11]. In contrast to our approach that relies on the similarity between dependency sets, `JDeodorant` relies on the concept of program slicing to select related statements that can be extracted into a new method. Our approach, on the other hand, is not based on specific code patterns (such as a computation slice). It is also more conservative to preserve program behavior (although it is currently restricted to non-contiguous fragments of code), and it relies on a scoring function to rank and filter recommendations.

There are other techniques to identify refactoring opportunities based, for example, on search-based algorithms [8], Relational Topic Model (RTM) [1], metrics-based rules [3], etc., that can be adapted to identify Extract Method refactoring opportunities.

4. Final Remarks

`JExtract` implements a novel approach for recommending automated Extract Method refactorings. The tool was designed as a plug-in for the Eclipse IDE that automatically

identifies, ranks, and applies the refactoring. Thereupon, the tool may contribute to increase the popularity of IDE-based refactoring tools, which are normally considered underused by most recent empirical studies on refactoring. Moreover, our evaluation indicates that `JExtract` is more effective (w.r.t. recall and precision) to identify contiguous misplaced code in methods than `JDeodorant`, a state-of-the-art tool.

As ongoing work, we are extending `JExtract` to be able to do statement reordering to uncover better Extract Method opportunities, as long as the modification preserves the behavior of the original code. Moreover, we intend to evaluate our tool with human experts to mitigate the threat that the synthesized datasets did not capture the full spectrum of Extract Method instances faced by developers. Last, we also intend to support other kinds of refactoring (e.g., Move Method).

The `JExtract` tool—including its source code—is publicly available at `http://aserg.labsoft.dcc.ufmg.br/jextract`.

Acknowledgments: Our research is supported by CAPES, FAPEMIG, and CNPq.

References

[1] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A. D. Lucia. Methodbook: Recommending move method refactorings via relational topic models. *IEEE Transactions on Software Engineering*, pages 1–26, 2014.

[2] M. Fowler. *Refactoring: Improving the design of existing code*. Addison-Wesley, 1999.

[3] R. Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *20th International Conference on Software Maintenance (ICSM)*, pages 350–359, 2004.

[4] E. R. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18, 2012.

[5] S. Negara, N. Chen, M. Vakilian, R. E. Johnson, and D. Dig. A comparative study of manual and automated refactorings. In *27th European Conference on Object-Oriented Programming (ECOOP)*, pages 552–576, 2013.

[6] V. Sales, R. Terra, L. F. Miranda, and M. T. Valente. JMove: Seus métodos em classes apropriadas. In *IV Brazilian Conference on Software: Theory and Practice (CBSoft), Tools Session*, pages 1–6, 2013.

[7] V. Sales, R. Terra, L. F. Miranda, and M. T. Valente. Recommending move method refactorings using dependency sets. In *20th Working Conference on Reverse Engineering (WCRE)*, pages 232–241, 2013.

[8] O. Seng, J. Stammel, and D. Burkhart. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *8th Conference on Genetic and Evolutionary Computation (GECCO)*, pages 1909–1916, 2006.

[9] D. Silva, R. Terra, and M. T. Valente. Recommending automated Extract Method refactorings. In *22nd International Conference on Program Comprehension (ICPC)*, pages 146–156, 2014.

[10] R. Terra, J. Brunet, L. F. Miranda, M. T. Valente, D. Serey, D. Castilho, and R. S. Bigonha. Measuring the structural similarity between source code entities. In *25th Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 753–758, 2013.

[11] N. Tsantalis and A. Chatzigeorgiou. Identification of extract method refactoring opportunities for the decomposition of methods. *Journal of Systems and Software*, 84(10):1757–1782, 2011.